Developing the Cooperative Mission Development Environment

Christopher Rouff
Goddard Space Flight Center
Code 522.2
Greenbelt, MD 20771, USA
chris.rouff@gsfc.nasa.gov

Mary Ann Robbert
Bentley College
175 Forest Street
Waltham, MA, 02154-4705, USA
mrobbert@bentley.edu

ABSTRACT
The Cooperative Mission Development Environment (CMDE) was developed to unite independent applications and databases into a cooperating tool set capable of sharing data. As missions developed stand-alone tools to meet immediate needs and to automate paper functions, a proliferation of development and automation tools came into existence. It soon became apparent that data in one tool could be used by another tool and a significant improvement in productivity could be obtained by combining the tools and sharing the data between them. This paper examines: how the independent systems arose, why the need developed to integrate the independent systems, the integration approaches considered, the process used for integration, how coordination between tool developers was handled, challenges faced during integration, plus current challenges, and future plans.

Keywords
Client/server, middleware, islands of computing, integrating databases

INTRODUCTION
The Cooperative Mission Development Environment (CMDE) is a collection of heterogeneous mission development tools that share data through a middleware layer. These tools were originally developed independently and over time were linked together to provide a single environment for sharing and processing data [6].

CMDE evolved in response to a situation common in many large enterprises. Multiple processes were being performed manually causing delays and duplication of effort. As processes were computerized, focus was on the solution of the immediate problems. User interfaces were designed to resemble traditional paper forms and off-the-shelf products were installed to meet specific needs, while no overall architecture for integration and growth was taken into account.

The Mission Operations and Data Systems Directorate (MO&DSD) at NASA also ran into this problem. The MO&DSD is responsible for developing ground systems for satellites. Over the years different divisions in the directorate developed software tools to help in the development of various ground stations. These tools were developed to perform a specific task and often without knowledge of the existence of other tools, or under time constraints that made it impractical to integrate them with other tools. The result was the development of islands of computing.

As developers used these tools across several missions, it soon became apparent that data in one tool could be referenced or used as a starting point by another tool. Linking data between tools would reduce the effort needed for reentering data, manually cross referencing data for status reports, or tracking how changes in one part of a system would effect parts in another.

This paper examines the process of designing and implementing an architecture that would integrate diverse tools in place with minimum disruption to operations. We describe: why the independent systems needed to be integrated, the integration approaches and tradeoffs considered, the process used for integration, how coordination between tool developers was handled, challenges faced during integration, plus current challenges and future plans.

THE NEED FOR COMBINING TOOLS
While developing new missions, units within NASA developed systems to meet their particular mission's immediate needs, including requirement definitions, tracking, discrepancy reports, and configuration

management. The lack of cross system and cross mission visibility caused overlapping systems and duplication of data and resources. The individual growth in many cases resulted in equal but separate systems as well as complimentary systems.

Since systems were developed in isolation, no consideration was given to compatibility. Separate systems were implemented on a variety of platforms. These were vertically constructed to solve individual mission problems. Client platforms included Mac's and PC's, while hosts varied across missions as well as within missions. The disparate database management systems installed on the servers raised the possibility of even more integrity problems in the system.

An example of a tool that was developed in isolation and whose data was later determined useful for other tools is the Requirements Generation System (RGS) [10]. RGS automates the development, editing, review, approval, and creation of requirements documents. The system provides a historical database of requirements to promote reuse of requirements across missions. Like many tools, RGS was developed as a two-tier client/server system [7]. This allowed users to access the database through their local workstation at a remote site and off-loaded much of the work from the database server machine to the client machine.

Several other development tools reference the same requirement names and numbers as RGS and could benefit from the client/server model. It was noted too that productivity could be increased if users could search for a requirement on-line instead of having to search through a paper report, which might be several weeks old and out of date. Also, it would save developers' time if they could be notified immediately when a requirement was changed. For example, when a requirement is changed, both testers using the test documentation system as well as someone writing a discrepancy report using the discrepancy reporting system need to know what changes were made to see if a test needs to be changed or if a discrepancy has been satisfied.

Other examples of increased productivity include producing reports for managers on what requirements have not been tested or how many outstanding discrepancy reports exist and to what requirements or tests they are associated. With separate database systems, producing either of these reports was very time intensive. By providing a link between the various databases, the data could be cross-referenced and reports generated automatically. In addition, the status of a project could be more closely monitored and developers could also be better informed of outstanding issues.

After it became apparent that an increase in productivity could be gained by linking or combining tools together, a Process Improvement Committee (PIC) was formed to investigate the feasibility of integrating the tools. The committee was made up of management and technical representatives for each of the tools, with a systems engineer or architect leading the meetings. Since there were initially nine tools being represented, there were often anywhere from 20 to 30 people at the meetings.

The initial goal of the PIC was to find a way to reduce the number of tools being supported, and to determine a way to combine or link the remaining tools together. The first set of meetings were devoted to presentations and demonstrations of the tools since most committee members did not know the details of all the tools. These demonstrations often become quite lively with each person extolling the benefits of their tool, why it should be used by everyone, and the part the tool played in NASA's overall mission. After the demonstrations, it became apparent that each tool filled a particular need or process. Because of this, it was determined that the number of tools could not be immediately reduced without having to go back to a manual process for some of the tasks.

Since none of the tools could be removed, the next step taken was to determine whether we should buy, redevelop, or link together the existing tools. Buying new tools was rejected because there were no commercial products that performed the same as the in-house developed tools (which is why they were developed) and the current set of users for each of the tools were reluctant, at best, to switch. Development of a whole new set of tools in a single environment was rejected for three reasons. The first being that it would take two to three years to develop a new suite of tools and developers needed something much sooner. The second was that the users would have to learn a whole new set of tools, which many would be reluctant to do. The third reason was the lack of funding for developing a completely new application suite given the time constraints.

This left us with the solution of linking the existing tools together. This solution had several advantages. Since users are often reluctant to switch applications, they could continue using the tools that they were familiar with (and sometimes helped develop) and there would be little or no learning curve or down time for learning yet another tool. In addition, linking the tools would take much less time than redevelopment, thus allowing an increase in productivity much sooner. This strategy also allowed new or commercial tools to be linked into the group as they became available.

COMBINING THE EXISTING TOOLS

Trying to integrate tools that were being developed and maintained across organizational boundaries had its challenges. The first challenge was how to do the integration. The second challenge was the logistics of integrating tools developed and being maintained in separate organizations with separate budgets and differing priorities for allocating resources to the integration.

Picking a Design for Integrating the Tools

We initially came up with the following possible designs for integrating the tools:

- distributed database, using the tools' current databases,
- centralized database, which combined the separate databases,
- combination of the first two (semi-distributed).

The first design required each of the tools to directly access the data in the other tools using a constellation of star configurations, as in Figure 1 (the acronyms in the figures are tools that would benefit from being linked together). In this design all tools were responsible for storing the relationship of their data to the data in other tools, as well as the address of the other tool's database. Each tool developer was also required to write an API to allow access to their data by other tools.

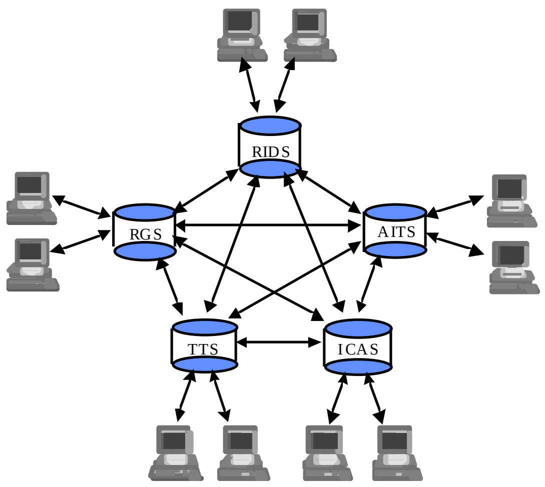


Figure 1: Constellation of star configurations.

The advantage of this technique was that the performance would be fast and the cost of development would be spread out over several departments. The disadvantages of this approach were:

- if one tool needed to change it's API, all tools would have to be updated at the same time, and
- if one server went down it would effect all the other tools that accessed the data on that server.

It was decided not to go with this approach because some of the departments were on limited budgets and might not be able to do the necessary upgrades. In addition, trying to update each of the tools in a coordinated fashion would be very difficult, since the tools and the authority to change them was distributed across departments. There were also concerns on security with many tools accessing each other's databases. Keeping track of who was authorized to access each database would have to be the responsibility of each individual tool. Keeping each tool updated with this information would be a large configuration problem.

The second design was based on the fact that there might be a department that may not have the budget to make any modifications to its tool, but the data stored in their database would be useful to other tools. This method would maintain a copy of all the tools' databases in a centralized database (Figure 2). This way the data was available to all tools yet each of the tools could maintain their own version. The central database would be updated on a batch basis since the data referenced in one tool was usually generated by another tool in a different part of the development lifecycle. Because of the large span of time in which the data was developed and the limited number of users, it was determined that real-time updates were not necessary. However, it was decided not to go with this approach because it would be the most costly and the hardest to maintain.

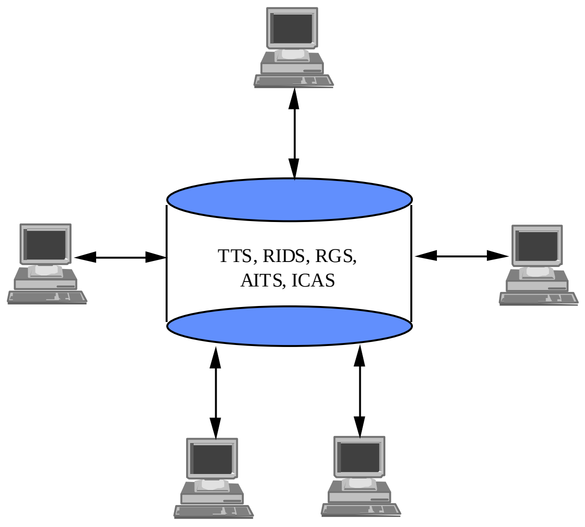


Figure 2: Centralized repository for all tools.

The third design combined the advantages of the above two and could be configured for flexibility and growth. A three-tiered client/server model was designed with a middle layer (middleware [1,3]) that all tools accessed first. This approach maintained a copy of only the relationships between data in a centralized database with pointers to the databases that maintained the real data. This still meant that each tool had to develop an API to access the middle layers, though this API would be smaller and less subject to change. From a security standpoint it was also an improvement. All security checking would be done through the centralized database and therefore remove the responsibility from the individual tools.

The middleware design also allowed each of the tools to have access to all inter-tool data relationships and the relationships could be changed without having to notify every tool, as would have been the case in the completely distributed solution. With the semi-distributed solution, the data reference in the central database would be marked as changed (with a time/date) and the other tools could check this periodically or when they referenced the data. This was a better solution than having to broadcast the change and verify all the databases got the message.

It was decided to go with the semi-distributed approach since it provided a good balance between the advantages and disadvantages of the fully distributed and centralized approaches. We liked the idea of not having to maintain a centralized repository for all of the tools, which would always be out of date due to the batch updates. The tool developers were also more comfortable from a security standpoint knowing that all data requests would be coming from the central database as opposed to many tools in the distributed approach. We also felt that tool developers would take the time to write the needed APIs for their tools if they were limited in scope and they knew it would result in more people using their data and tools.

Logistics of Integration

After we determined the design, the new challenge became how to coordinate all the developers of the tools. Since all the developers were responsible for developing some APIs to allow the central database to access their data, we could not test our link database prototype until at least two complimentary tools developed APIs.

We immediately came across conflicting schedules between the developers and the priority each of the departments had in implementing the integration of the tools. Each of the development groups were usually under time constraints to release a new version of their software, or did not have the funding to implement the modifications, or they were not convinced the concept had merit and were not willing to allocate resources. Time constraint concerns were sometimes valid, especially when a groups' data was more important to other tools than the other tools' data was to a group. People were also reluctant to make modifications when no other tools had made theirs, or until the concept was proven.

Due to these conflicting goals, differences of opinion on implementation strategies, priorities, budgetary constraints, and the need to coordinate development personnel across departments, it was decided to locate five of the tools in a single department under two managers. This allowed the developers to better coordinate development priorities given time constraints and limited budgets.

IMPLEMENTATION OF CMDE

Once the design had been chosen (semi-distributed) and the logistics of modifying several tools at once were addressed, we needed to decide whether to do the middleware implementation with a commercial off-the-shelf product or write custom code.

Picking an Implementation Strategy

After doing some tool studies, we found that there were middleware tools commercially available that could be used to do the integration (such as [4,8]). Unfortunately, no single tool we found could provide the required cross platform access and guarantee the needed functionality in the given time frame and within our budget. The risk also existed that even state-of-the-art commercial middleware services might not keep pace with the changing technology [3].

Another problem we found with off-the-shelf products is the lack of in-depth understanding required of developers. The interfaces provided in the tools are abstract to a level where it can be difficult to determine when errors occur whether they are in the middleware product, the database, the client, or the design. Technical assistance, when available from the vendor, is generic and does not always solve specific implementation problems. Since our initial proof of concept did not require all the features included with commercial middleware, it was determined that constructing our own middle layer would clarify design issues, ensure local control, allow for better maintenance, and still give us the option of converting to a commercial product down the road [9].

The middle layer was designed as independent modules that can be added or modified with minimum impact on other modules. Because of this design and the need to get CMDE out to the users who would benefit the most, as soon as possible, it was decided to do the integration in an incremental fashion. A single development program, based on our budget, would require several years before users could start achieving benefits. There is a vision of a mission development tool suite existing in a single environment, but the path to obtain that goal is not a straight one, but an incremental one. The goal is to link tools together one at a time. This demonstrates the tool viability and the benefits derived from the product if budgets should change and development must be slowed or stopped.

Four stages of integration were planned. The first stage was to export data so others could import it, in the second stage tools display data stored in other tools through a tight coupling, the third stage provides a means to display the data through a common middleware, and the fourth stage is the integration of the tools and databases into a common mission development environment.

The first stage of integration actually was unplanned and happened when it became apparent that other tools could use the data in RGS. An example of this was a testing tool that required the user to type in the requirement text that they were testing. To accommodate the testing and other tools, an export feature was built into RGS. This allowed other tools to import the text of requirements into their tools or word processors. However, if changes are made to the data in RGS, the tools that imported it are not notified since there is no mechanism to keep track of other tool's imports.

The second stage of integration allows users to display data in one tool while working in another. This was needed because one tool required the users to input a requirement number when entering a discrepancy in an automated discrepancy system. Users needed the ability to browse or search through the RGS to find a requirement as they were writing a discrepancy, instead of reading through a paper version of the requirements. This is currently implemented through a web-based read-only browser which displays the requirement and lets the users copy and paste requirement numbers or text of the requirements into the discrepancy tool.

The third stage of integration, which has partially been completed, was achieved through the addition of the middle layer to provide sharing of data between the tools. Figure 3 shows the architecture. The middleware performs the needed security checks and stores the links between the data with pointers to the actual data in the appropriate tool. An example of this is with RGS. A second tool, the Test Tracking System (TTS), needs to have each test reference a requirement that it is testing. The integration is sufficient to allow the testers to immediately view the requirement that the test is being written for and also allows them to be notified when a requirement has been changed. Users will no longer be required to manually import all the requirements from RGS and then edit out the requirements they are not interested in.

The fourth stage of integration, combining the existing databases into a single mission database, is preferable from the current semi-distributed implementation because of speed and easier access. A decision has been made to transfer CMDE from NASA to industry, so this stage of the integration will be done by another party.

There are several challenges in integrating the existing databases. They are all organizational, not technical. The primary challenge is getting organizations to buy into giving up some of the control they now have through maintaining their own databases, for the benefit of faster and easier access to all of the other mission data. Another challenge is maintaining confidentiality over data. If a project does not want anyone else to see their data they currently can limit access to their group, since they are in charge of it. Yet another challenge will be for some tools to obtain the funds to do the necessary conversions.

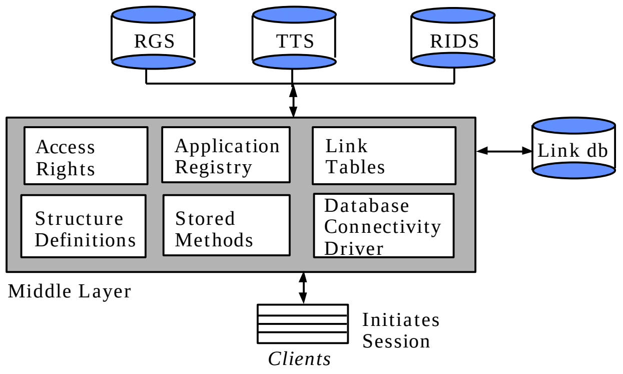


Figure 3: Semi-distributed solution.

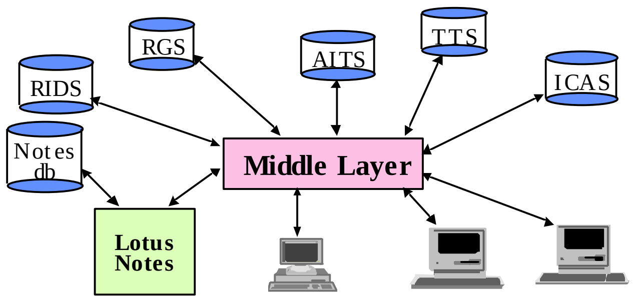
FUTURE ENHANCEMENTS

Plans for future expansions or modifications include: introduction of workflow, incorporation of the World Wide Web, plus other foreseeable enhancements.

Adding Workflow

It has been demonstrated that an integrated product such as CMDE reduces the quantity and length of required meetings and time to complete tasks [5]. Further reductions in meetings and a speedup in task completion time can be accomplished through the integration of workflow support. Interactive discussions, with everyone's contribution visible to all parties, would simulate group meetings. Requirements could be negotiated on line and modifications agreed upon without having to coordinate meetings. A workflow product could correctly route documents, track and verify approvals, and send reminders for incomplete tasks. Collaborative work is required throughout development but especially in design and test phases.

A groupware product, such as Lotus Notes, would provide communication, collaboration and coordination capabilities. As a single product, Notes provides the synergy between messaging (e-mail), shared database (conferencing), and development (workflow). A groupware product can be added to the model with no significant changes (see Figure 4). Users enter through the middle layer, have their access rights verified and are connected to the appropriate Notes database the same as to any other database. The initial introduction would add Notes as a workflow tool and a group discussion facilitator.



Expanded Architecture

Figure 4: Expanded three-tier architecture.

World Wide Web

Accessibility to the world wide web is becoming universal. Web use can decrease cost and permit concurrent access to information. RGS has developed a web browser, using Cold Fusion, which allows read only access to all RGS mission requirements. Subsequent versions will require users to enter through CMDE which will check for proper clearances and define links. A URL method will access a mission browser through Netscape, allowing any authorized user to review mission requirements without additional client software. Added security, integrity controls and recovery procedures stored in the middle layer will eventually permit the user to update and approve requirements through the web.

Other Enhancements

Additional connectivity is envisioned to include external files, and external programs, including real time programs. Future versions could contain search engines, with the capability of searching data within external files and databases, to retrieve matching requirements. Ultimately, links could be established in response to natural language requests.

LESSONS LEARNED

We learned several lessons while developing CMDE. The most important is how to deal with constantly changing technology. The second is how to deal with the logistics of modifying tools across organizational boundaries.

Constantly Changing Technology

One of the biggest challenges we faced in developing CMDE was (and still is) the constant change in technology. Commercial tools are constantly changing and new ones are becoming available. After evaluating a set of commercial tools that might do the integration, another tool or new feature to an existing tool would be announced that held the potential for solving our problem without having to resort to custom software. This was particularly true when we where presenting our designs to the Process Improvement Committee. Invariably, after doing a presentation, someone would say that they had heard of a new tool or new version of an existing tool that might solve all of our problems. By the time we evaluated the new tool and came to the conclusion that it would not solve our problem, another one became available, which would put us back to square one. Also, in some cases it was just promises of new features of a tool to come in the next one or two months.

In this situation, where there are many people watching from several organizations, it is very easy to never come up with a solution since it is often thought that the best solution needs to be found, and there constantly appears to be another better solution on the new horizon. This situation is similar to people who never buy a personal computer because in three or six months new technology will become available making what they buy today obsolete or overpriced. We also got into this situation where we were spending an inordinate amount of time evaluating or waiting for new technology that might solve our problem and no progress was made to solve our current problem. We finally had to draw the line and say we were not going to evaluate any more technology. We instead decided to provide a prototype that was capable of integrating new products as they developed.

Combining Tools Across Organizational Boundaries

Organizing the integration of tools that were developed and being maintained across organizational boundaries provided more challenges than any of the technology itself. Providing a committee of the managers and tool developers helped people see the need for integration and got buy-in from the required parties. When it came to the actual design of how to do the integration, the committee approach proved to be more a hindrance than help.

Providing periodical updates on the status of the project to the other organizations was essential for ensuring continued support. If we did not do this, after two or three months the other organizations thought the effort had been abandoned. On the other hand, having too many meetings made us spend excess time developing presentations and chasing new technologies that someone had heard about and thought we should look into before proceeding any further. When looking into new technologies, we found it was best to let people know how long it would take and how much we would have to delay the current work. Otherwise, we found that people greatly underestimated the time it took to investigate a new technology.

CONCLUSION

We have found that integrating stand-alone tools into a shared environment is feasible. In the Mission Operations and Data Systems Division stand-alone tools were often developed or purchased to solve local automation problems producing several stand-alone systems. As these tools developed and their use spread, duplication and overlap became evident. An evolutionary integration process based on a three-tiered architecture was designed to join the existing tools with minimal disruption of on-going projects. This model, designed for future development allowing for changes in technology and processes, can be replicated in other domains where separately developed tools require integration.

Our project goal was to automate as much of the mission development process as possible with the quickest return possible. We determined code written in house would be best to unite our diverse systems for prototype development. This has not only provided us with a low cost working model but also an understanding of the technological issues being addressed. We are beginning the evolutionary integration process realizing the technological problems and more aware of users needs.

The success of this and similar projects is dependent on usage by all parties participating in the project. We elected to use an evolutionary process adding tools one at a time. This makes the measurement of satisfaction difficult. As the first tools were integrated users expressed satisfaction and made suggestions. These suggestions are, where possible, being incorporated before the next tool is integrated. This improves the product and increases user ownership of the system, but adds another variable to the design process making it even more challenging to evaluate.

REFERENCES

1. Bauer, M.A., et al. A Distributed System Architecture for a Distributed Application Environment. IBM Systems Journal, 33, 3, 1994, 399-424.
2. Bernstein, P. Middleware: A Model for Distributed System Services. Communications of the ACM, 39, 2, February 1996, 86-98.
3. Greenbaum, J. But you don’t get it: Middleware. LAN Times, August 28, 1995, 73-76.
4. Kramer, M. Tuxedo System 5: Tools for Building Three-Tier Applications. Available at http://www.psgroup.com/news/1995/2nc195d.htm.
5. Mandl, D., et al. Overcoming MO&DSD’s Learning Disability to Building Ground Data Systems "Faster, Better, Cheaper". Internal NASA Publication, July 1996.
6. Robbert, M.A., Rouff, C., and Burkhardt, C. A Cooperative Mission Development Environment for Crossplatform Integration. In Proceedings of SAC’97: ACM 12th Annual Symposium on Applied Computing, February 28 - March 1, 1997. San Jose, California.
7. Sinha, A. Client-Server Computing. Communications of the ACM 35, 7, July, 1992, 77-98.
8. SQL*NET. Available at http://www.oracle.com /products/networking/htm/stnd.sqlnet.html.
9. Umar, A. Distributed Computing and Client-Server Systems. Prentice Hall. 1993.
10. User’s Guide for the Requirements Generation System (RGS). National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland, June 1995. Available at http://stellar.gsfc.nasa.gov/csdev/ug40/htframe.htm.